Towards model-based integration of component-based automotive software systems

Johannes Schlatow, Mischa Möstl
and Rolf Ernst
Institute of Computer and Network Engineering
Technische Universität Braunschweig
{schlatow,moestl,ernst}
@ida.ing.tu-bs.de

Marcus Nolte, Inga Jatzkowski
and Markus Maurer
Institute of Control Engineering
Technische Universität Braunschweig
{nolte,jatzkowski,maurer}
@ifr.ing.tu-bs.de

***Abstract*—The increasing complexity of automotive software systems and the desire for more frequent software and even feature updates require new approaches to the design, integration and testing of these systems. Ideally, those approaches enable an in-field updatability of automotive software systems that provides the same degree of safety guarantees as the traditionally lab-based deployment. In this paper, we present a layered modelling approach that formalises the integration procedure of automotive software systems using graph-based models and formal analyses.**

I. Introduction

The current trend towards autonomous driving technologies gradually pushes the complexity of automotive software systems. In particular, the development of advanced driver assistance systems (ADAS) not only requires more sophisticated and complex sensors for environment perception but also the application of state-of-the-art software design methodologies that help to mitigate side effects and to reduce interdependencies. In other words, separation of concerns is a major objective when it comes to the design of complex software. This results in a more interface-oriented approach which ideally reduces the scope that must be checked for side effects. We can observe this trend as a growing interest in component-based software and service-oriented architectures in the automotive domain [1], [2]. In the general case, however, non-functional requirements – such as real-time, safety or security – cannot be verified on an interface-/service-oriented basis but instead require an in-depth understanding of the entire system on different layers. This is due to the fact that multiple (software) components execute on the same platform (operating system/system on chip/processor/etc.) and may therefore influence each other not only via software interfaces but also on other system layers. For instance a flawed device driver may easily impede the timing of the application software. This becomes increasingly relevant when we look at the trend towards high-performance platforms such as the zFAS[1] from Audi or the Drive PX[2] from NVIDIA.

On the layer of the operating system, which connects the application software with the hardware, microkernels are a well-known concept that provides a tamper-proof isolation of software components, which bases on fine-grained and flexible access control. In our point of view, this closes the gap between rather static systems as in OSEK/AUTOSAR, and very dynamic and open systems such as Linux. The fine-grained control of privileges (at run time) makes the interfaces explicit, and enables containment of software errors and security leaks. We consider these systems as having a fixed configuration that, although changeable at run time, must only be modified in a controlled manner. The challenge in this regard consists in finding a composition of software components (and their privileges) that implements the desired functionality and fulfils all given requirements.

In the scope of the collaborative research project Controlling Concurrent Change (CCC)[3] funded by the German Research Foundation (DFG), we develop model- and component-based methods for solving the integration and update problem of critical embedded systems such as automotive vehicles. Here, we focus on the integration problem which consists in finding new system configurations rather than modifying/updating only particular parts. Note that, from a modelling perspective, this is a prerequisite for considering later changes as a complete system understanding must be available to safely exclude any side-effects. For this purpose, let us have a look at the traditional V-model development process. In this model, the left branch of the V represents the top-down approach that gradually refines the design and implementation from system layer over subsystems to single software components and modules. The right branch of the V then models the corresponding integration and testing on the different layers in a bottom-up approach. Our ultimate objective is to automate the integration phase of this process, which will enable in-field updates as the vehicle can thereby perform the essential integration steps by itself. Note that this model, however, only illustrates the very abstract nature of the development process without design iterations or product variants. We also observe that this process typically tailors the design to a particular (hardware) platform which is chosen in the beginning. In order to automate the integration phase, we have to assume some modifications to this process: We assume the design phase is platform-independent and results in a library of reusable and tested hardware/software components with clearly defined interfaces. The integration phase then starts with combining these components to achieve the desired functionality on the

[1] http://audiblog.co.uk/featured-2/zfas-the-brain-behind/
[2] http://www.nvidia.com/object/drive-px.html
[3] http://ccc-project.org

https://doi.org/10.24355/dbbs.084-201803221525

target platform. This phase is formalised and augmented with (formal) verification techniques based on additional model information extracted from the design phase. As a result, testing may only be required as a means to validate the adherence of a component's implementation to its model. In this regard, the automated integration must replace design decisions on the upper layers which are often based on experience and expert knowledge. In this paper, we present our top-down approach which bases on formal models for the different layers and their composition. More precisely, we use graph-based models for each layer and mapping relations between the layers.

The following section summarises the automotive application scenario for which we develop and evaluate these methods in CCC. In Section III, we present our meta-model for the automated integration of these applications as it evolved from our experience within CCC and briefly summarise some practical implementation details in Section IV. Furthermore, we will elaborate in Section V on the challenges and opportunities this approach involves, before we have a look at the related work in Section VI and finally conclude our findings in Section VII.

II. Application Scenario

In the CCC project, the TU Braunschweig Institute of Control Engineering contributes two experimental vehicles which act as demonstrators for the developed run-time environment: A 1:5 model vehicle (**Modul**ar **X**-by-Wire, MAX) and a full-scale x-by-wire vehicle (MOBILE). Both vehicles currently act as platforms for research in E/E-systems and vehicle dynamics. MOBILE features four close-to-wheel electric drives ($4 \times 100\,\mathrm{kW}$), individually steerable wheels, and electro-mechanic brakes [3]. A similar actuator topology has been implemented for MAX. Both vehicles feature a FlexRay communication backbone for inter-ECU communication and additional CAN bus interfaces, which are, e.g., used for communication with sensors and actuators in the full-scale vehicle. The implemented applications on the vehicles can be divided into *control applications* and *automated driving functions* based on environment perception algorithms. In order to demonstrate the applicability of the CCC approach, several updatable applications have been selected for implementation on the research vehicles:

A *cruise-control application* serves as a basic example with soft real-time requirements, considering the rather inert lateral dynamics of the research vehicle. Three different variants are implemented. Besides a basic variant, two extensions feature a speed-limit detection and vehicle-to-vehicle communication, both imposing limits on the controlled velocity of the vehicle.

The *force-feedback application* provides haptic feedback at the steering wheel about the road surface to the driver. It serves as an example for a more time-critical application, due to the higher dynamics of the feedback actuator at the steering wheel. Application updates are limited to parameter updates so the desired strength of the feedback is configurable.

The *stability control* keeps the vehicle controllable for the driver performing yaw-rate and slip control. As it needs to be able to control the vehicle at the limits of handling, the controller is sensitive to varying cycle times. The application is intended to provide updatable control strategies, ranging from differential braking to advanced strategies utilising all available actuators in the vehicle. The stability control also provides a basis for the implementation of automated driving functions in the CCC framework. For this, the application will be extended to a use case of automated obstacle avoidance. The resulting trajectory-following stability control will use data acquired from multiple environment sensors (three LiDAR scanners, a radar sensor, and a camera). From this data a map of the static environment is created which provides the basis for a model-based trajectory planning utilising all actuators (particularly all-wheel steering) for maximal manoeuvrability.

III. Cross-Layer Integration

As motivated in the introduction, we present a cross-layer modelling approach for the automated integration of software components in automotive systems. In this section, we formalise our model layers as directed graphs and introduce their top-down pattern-based transformation.

Definition 1 *(directed graph) A directed graph $G = (V, A)$ is defined by a set of vertices V and directed edges (arcs) $A \subseteq V \times V$, i.e. $\forall a = (v, u) \in A : v \neq u \wedge v \in V \wedge u \in V$.*

Definition 2 *(arc splitting) An arc splitting of a directed graph $G = (V, A)$ is a directed graph $G' = (V \cup \{s\}, (A \setminus \{(v, u)\}) \cup \{(v, s), (s, u)\})$ which contains an additional intermediate vertex s on arc (v, u). We denote the result from any iterative application of an arc splitting on G as an* arc-split graph *from G.*

Definition 3 *(pattern-based transformation) A pattern-based transformation transforms a graph $G = (V, A)$ into a graph $G' = (V', A')$ such that $\forall v \in V : \exists v' \in V'$ and $\forall a$ in $A : \exists a' \in A'$, i.e. every vertex/arc in G has at least one corresponding vertex/arc in G'. The transformation is performed by replacing every $v \in V$ with a pre-defined pattern $P(v)$. A pattern $P(v) = (V_v, A_v, O_v, I_v)$ is a graph with nodes V_v, internal arcs A_v, and output/input nodes O_v/$I_v \subseteq V_v$. The transformation patterns are applied such that $V' = \bigcup_{v \in V} V_v$ and $A' = \bigcup_{(v,u) \in A} (v', u') \cup \bigcup_{v \in V} A_v$ with $v' \in O_v$ and $u' \in I_u$, i.e. G' comprises all the nodes and internal arcs from the patterns as well as transformed arcs from G, which connects an output vertex with an input vertex of the corresponding patterns*[4].

The additionally required model information – such as interface compatibility, service dependencies, transformation patterns – is stored in a so-called *component repository*, which we will describe in more detail in Section IV.

In order to illustrate the tasks performed by our automated integration procedure, we introduce an exemplary use case which generalises the common sensor-decision-actuator scenario in vehicular systems based on a simplified cruise-control application. Note that we structure the remainder of this section top-down into a functional view, a component view and a thread view. A view combines several layers that

[4] or vice versa, depending on the semantics of the transformed arcs

https://doi.org/10.24355/dbbs.084-201803221525

are similar in their semantics but represent different steps in the integration procedure. We will further elaborate how these layers are reflected in our use case (cf. Figure 1).

A. Functional View

We split the functional view into two layers: the platform-independent functional architecture and the platform-specific communication architecture. The primal entities of both layers are functional blocks, which form the set of vertices. The arcs represent data dependencies between the functional blocks.

1) Functional Architecture Layer: We consider the functional architecture as the input to our automated integration procedure. Knowing the dependencies on the functional level additionally enables a thorough assessment of the impacts of any degrading or deviating performance on the functional safety which may rely on certain assumptions on the platform or the implementation. We base this assessment on skill/ability graphs, which we briefly summarise in Section V. Here, we only assume that such an assessment exists – either automated or manual – and that it imposes additional requirements on the integration procedure.

Definition 4 *The* functional architecture graph $G_{func} = (V, A)$ *is a directed graph with functional blocks* V *and their data dependencies* A*. An* $a = (v, u) \in A$ *denotes that* u *depends on data from* v*.*

Data dependencies on this layer indicate that there must be some communication between the connected blocks in order to exchange the corresponding data but do not specify how this data is exchanged. We refer to this as *semantic compatibility* that must be guaranteed from the functional architecture.

The functional architecture of the cruise-control application shown in Figure 1a consists of a functional block which performs speed control according to the velocity selected by the driver via a human machine interface (HMI). This block takes inputs from sensors to measure the actual velocity of the vehicle and commands an appropriate torque to the drive train. We further assume that wheel speed sensors and an inertial measurement unit (IMU) are present to measure velocity.

2) Communication Architecture Layer: This layer starts tailoring the given functional architecture to a particular platform consisting of multiple CPUs and/or electronic control units (ECUs). From our experience, it is beneficial to include platform- and implementation-specific considerations early on as this significantly narrows the corridor of feasible configuration in the design space. This includes a distribution of the functional blocks across the ECUs based on model information on their compatibility. Another aspect is to decide for communication interfaces between the ECUs, which we explicitly model in this layer by inserting so-called proxy blocks between (connected) functional blocks that are mapped to different ECUs. These proxy blocks serve as a place holder for any inter-ECU communication mechanism that guarantees that both functional blocks can actually exchange data (reachability problem).

Definition 5 *A* communication architecture graph $G_{comm} = (V, A)$ *is an arc-split graph from the functional architecture graph with the only difference being the semantics of the arcs: An arc* $a = (v, u) \in A$ *not only denotes the semantic compatibility of the connection between* v *and* u *but also that* u *is reachable from* v*, i.e.* u *and* v *are mapped to the same platform component (ECU).*

We consider a publisher-subscriber mechanism for inter-ECU communication that allows to publish data at defined points in time. This mechanism does not resort to strict time triggering such as TDMA because the computation time is arbitrary but constrained, i.e. as long as it finishes before the specified publishing point the publishing time can be enforced, which keeps the communication timing deterministic. We refer to this as logical execution time (LET) communication [4]. This allows us to decouple the communication between ECUs from their implementation (internal architecture, operating system, run-time environment) and thus to take early decisions on particular time budgets.

This layer is illustrated in Figure 1b, which depicts the mapping of the functional blocks to the platform components, and shows the insertion of the aforementioned proxy blocks.

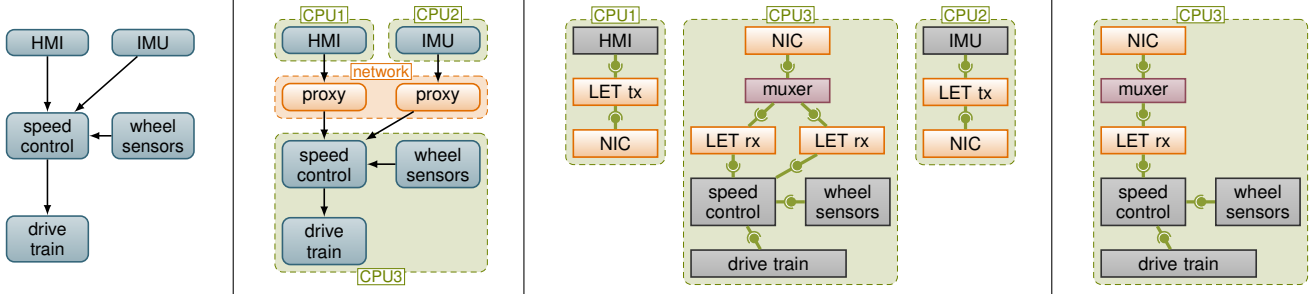
B. Component View

The component view starts including implementation-specific aspects based on the given (functional) communication architecture, which is first converted into the component architecture by a pattern-based transformation. In this case, a pattern describes how a functional block is implemented by a network of software components. By deciding between alternative patterns, different implementations can be chosen subject to non-functional requirements. The component architecture layer also deals with interface compatibility by inserting library components if required.

Note that the component architecture may contain the same component at several places. As we use service-oriented interfaces between components, i.e. a component can provide a service to various other components, not every component needs to be instantiated multiple times. Moreover, device drivers – which are also software components in the microkernel approach – must only exist once as they require exclusive access to the corresponding hardware device. This is addressed by the component instantiation layer, which reduces the component architecture to a minimum set.

1) Component Architecture Layer: In contrast to the upper layers, the vertices on this layer model software components whereas the arcs denote their connections via service-oriented interfaces that describe the dependency of a service provider and its client(s). The objective of this layer is to derive an implementation-specific and complete model of the software architecture for a given communication architecture. As a first step, this involves a pattern-based transformation of the latter (i.e. its functional blocks) into application components. Secondly, it addresses the compatibility of the connected interfaces and their cardinality, i.e. the maximum number of clients. For this purpose, we insert available library components that either perform a translation between different interfaces (*protocol stack*) or act as a multiplexer of a service

https://doi.org/10.24355/dbbs.084-201803221525



(a) functional architecture

(b) communication architecture with platform components (CPU1-3 and network)

(c) component architecture showing the pattern-based transformation of the proxy blocks into NIC and LET tx/rx components (orange) and the insertion of a multiplexer

(d) component instantiation (only CPU3)

Figure 1. Illustrative cruise-control example on the modelling layers of the functional and component view. (HMI = human machine interface, IMU = inertial measurement unit, NIC = network interface controller, LET = logical execution time)

interface (*muxer*), both without modifying semantics of the exchanged data. Similar to the semantic compatibility dealt with on the functional layers, we refer to this as syntactical compatibility which is ensured by this layer.

Definition 6 *The* component architecture graph $G_{comp} = (V, A)$ *is an arc-split graph from the pattern-based transformation of* G_{comm}*. The vertices* V *represent the software components (i.e. application and library components). An arc* $(v, u) \in A$ *denotes that* u *provides a syntactically compatible service to* v*. Note that the arcs in this graph may therefore have a reverse direction to the corresponding arcs in* G_{comm}*.*

Figure 1c depicts this layer for our illustrative use case. The functional blocks have been transformed into software components. For better readability, we assumed a simple transformation into a single component for most of the functional blocks. Each of the proxy blocks, however, was transformed with a more complex pattern: The *LET tx* component transmits the data from the HMI to the *LET rx* component on CPU3. For this purpose, both components are connected to a *NIC* component, which represents the device driver for the network interface controller (NIC). Moreover, as this device driver must only have one client, a multiplexer component (*muxer*) has been inserted on CPU3.

2) Component Instantiation Layer: As mentioned above, the purpose of this layer is to reduce the component architecture to a (minimal) set of component instantiations under cardinality constraints, i.e. the maximum number of instantiations of a component on a particular CPU or ECU. The component instantiation graph has similar semantics as the component architecture graph G_{comp} but models the actual instantiation of software components.

Definition 7 *The* component instantiation graph $G_{inst} = (V', A')$ *models the instantiations of software components in* $G_{comp} = (V, A)$ *such that for every vertex in* V *there is a vertex in* V' *that refers to an instantiation of the same software component. Similarly, every arc in* A *can be related to an arc in* A' *that connects the appropriate instantiation.*

Yet, this reduction is not a trivial task as it must ensure that the paths between sources and sinks in the component architecture are preserved. It must also respect that components executing on different platform components must not be mapped to the same instantiation. As presented in [5], this can be approached by constraint solving techniques that will respect all these constraints and can also be combined with objective functions to minimise the number of instantiations.

An example of a subset of this layer is shown in Figure 1d. On CPU3, the two *LET rx* components could be merged into a single instantiation that implements both communication channels to the HMI and IMU. In this small example, this actually renders the multiplexer component superfluous, which needs to be taken care of separately. In realistic use cases, however, this is rarely the case as there will often be more components accessing a shared service provided by a device driver. We omitted the model for CPU1 and CPU2 in this example, as they are the same as in the previous layer.

Note that our approach of separating the component architecture from the instantiations can also be applied to classic automotive software models in which runnables are the atomic entities of the software architecture. Multiple runnables are executed within the same periodically executed software component. We can therefore model the interaction (communication) between the runnables on the (component) architecture layer and their mapping to the actual software components on the (component) instantiation layer.

C. Thread View

As previously stated, the integration procedure for automotive software systems is subject to timing requirements among others. Please refer to the AUTOSAR Timing Extensions [6] for a quite complete summary of relevant timing constraints. We introduce the thread view in order to perform a timing analysis that formally verifies the adherence to the given requirements. More specifically, the thread communication layer models the interaction between software components in more detail – comparable to sequence diagrams – such that a proper timing model can be derived as, e.g., in [7]). Yet, as the timing analysis has previously been addressed in [8], we only focus on the model transformation from the component instantiation graph in the scope of this paper.

1) Thread Communication Layer: We consider single- and multi-threaded software components. A thread is a sequentially

https://doi.org/10.24355/dbbs.084-201803221525

executed part of a software component that may communicate with other threads. Multiple threads can run concurrently.

We distinguish three different thread communication mechanisms: implicit communication, synchronous inter-process communication (IPC) and asynchronous messages. Implicit communication is dominant in control systems where data is sampled periodically, i.e. it is read from shared memory following a time-triggered activation. Synchronous IPC and asynchronous messages correspond to the prominent mechanisms in microkernels. They also reflect the different semantics modelled by sequence diagrams: the procedure call, which blocks the caller until the callee replied, and the notification, which triggers the receiver but does not block the sender's execution. As was mentioned in Section III-A2, we apply the LET paradigm between inter-ECU communication. Due to the fact that LET is an implicit communication, we only need to consider the thread communication on a CPU/ECU basis. W.r.t. the verification of timing requirements, this allows us to apply a response-time analysis as shown in [8] to derive upper bounds on local response times and use these with an end-to-end latency analysis for the LET paradigm.

Definition 8 *The* thread communication graph $G_{th} = (V, A)$ *is derived from* G_{inst} *purely by a pattern-based transformation. The patterns describe the activities of a software component in relation to its interaction with the connected components. As a result, the vertices* V *represent the activities and the arcs* A *model the trigger dependencies between the activities, i.e. an arc* $(v, u) \in A$ *denotes that* v *activates* u *with the associated communication mechanism.*

A detailed description of how these patterns can be specified for a software component and its interfaces is found in [9].

IV. Implementation

In this section, we briefly describe the most relevant implementation details of our approach; a more detailed account of this is presented with our case study in Appendix A. As mentioned earlier, the model information is stored in a component repository on a per-component basis. More precisely, the information is specified in a structured way using XML, which can be easily extended and thus suits the agile nature in our research unit. Due to space limitations, we omit syntactical details and rather focus on what (essential) information we specify for each component in a more abstract way. First, we specify the platform requirements of a component to capture the platform compatibility. In addition, a component has provisions and requirements that either relate to the functional or the component view and which describe the data/service dependencies and their compatibility. The provided services are annotated with the maximum number of clients as a cardinality constraint. A component providing a function is classified as an application component and implicitly specifies a possible transformation pattern for this function. The remaining components are classified either as a *proxy*, a *muxer*, or a *protocol stack*. We also capture whether a component is a singleton (cardinality ≤ 1) or not. Moreover, we specify functional blocks either as alternative sets of pre-connected components or single components. On the one hand, this provides the transformation patterns from the functional to the component view. On the other hand, it also circumvents transitive application of arc splitting as we can pre-define complex scenarios into a functional block that, e.g., specifies a protocol stack.

W.r.t. graph-based data structures, we emphasise that we are annotating the arcs and vertices with model information during the integration procedure such as the mapping decisions or the services to which an arc relates. For this purpose, we make use of the LEMON graph library[5], which provides a good support for these annotations.

For the in-field integration, we resort to a separation of a model domain, which implements our model-based methods to find valid system configurations, and an execution domain, which runs the configuration that was selected and fixed by the model domain. Please refer to [10] for more details of this architecture w.r.t. self-assessment capabilities.

V. Discussion & Future Work

We now extend our scope and discuss the benefits and challenges that arise from our aforementioned model. In the preliminary section, we only focused on the structural properties of the model and a limited set of views. One important aspect that we omitted thus far is how design decisions are made based on additional information during our automated integration procedure. More precisely, a *design decision* must be made on a certain layer if there are multiple options such as an alternative mapping to platform components or a different transformation pattern. For this, we incorporate so-called *analysis engines* that evaluate the model layers (or a selection thereof) at particular steps in the integration procedure. These engines basically serve two purposes: On the one hand, they can incorporate additional information – e.g. from orthogonal views or lower layers – in order to guide the decision-making either heuristically or holistically. As a result, they replace the *expert knowledge* and experience which is often required to make good design decisions (possibly on incomplete information) early in the design process. On the other hand, the analysis engines can implement established or novel analyses in order to perform formal admission tests so that they can give formal guarantees on whether the requirements and constraints of a particular view are satisfied by the current model. This design-space exploration may therefore involve backtracking if a design decision on an upper layer results in an infeasible model on a lower layer, which may lead to design iterations. We are currently working on methods that reduce the design iterations and thereby increase the efficiency of this exploration.

A major benefit that we see in our approach is its extensibility by additional views in order to model and verify requirements on, e.g., security, safety, availability. Such concerns can, in general, not fully be addressed within a functional or logic architecture alone. In the scope of this

[5]http://lemon.cs.elte.hu

https://doi.org/10.24355/dbbs.084-201803221525

paper, we want to focus on (functional) safety aspects, which we approach by a compositional approach of dependency modelling/analysis on the functional and platform levels. On the functional level, the concept of ability and skill graphs [11] has been proposed for supporting system design and run-time monitoring of automated vehicle systems. During design time, the system is composed of platform-independent skills which are required to fulfil the system's task. Abilities are preconditions for executing the corresponding skills and can thus also be attributed with a level of run-time performance, which dictates platform requirements or can be used as an input for monitoring the system. On the platform level, we perform a cross-layer analysis of dependencies to expose any implicit dependency between functionally independent components due to platform sharing [12]. W.r.t. safety aspects, dependency analysis enables identifying points of possible interference, e.g. between functions that must be separated according to the functional safety concept and the skill representation.

VI. Related Work

As far as model- and component-based development of embedded systems is concerned, there exist several tools and frameworks such as MARTE[6], EAST-ADL[7], or Rubus [13]. Most of them, however, aim at the *assisted* development of *statically configured* systems and thus do not suit fully-automated integration. EAST-ADL is an architecture description language that focuses on capturing automotive electronic systems through an information model on different levels of abstraction. The engineering flow behind this meta model is not intended for continuous integration efforts but rather focuses on variant management as known from classical approaches. Similarly, MARTE is an UML2 profile which aims at the description of timing properties of models specified, e.g. in EAST-ADL. However, both are only standard description such as e.g. AUTOSAR but not a concrete tool itself. The Rubus approach [13], for instance, pursues a similar strategy of modelling different views and their transformation in a layered manner for the design, analysis and code generation of vehicular software systems. In contrast to our approach, Rubus – as an all-in-one solution with a strong focus on the real-time aspects – is more mature and practical but also makes more restrictions such as requiring its own operating system. Furthermore, as it starts on the component level as the view of the development team, it does not accommodate automated changes on this level. Instead, we try to build a compositional layered model such that it is extensible in vertical and horizontal direction in order to incorporate additional views on the system such as (functional) safety, reliability, or security. Moreover, as we only focus on the automated integration procedure our models can be more abstract as they rely on certain properties that can already be checked in detail by the preliminary design process (e.g. interface compatibility). Similar limitations hold for other approaches, on which we cannot elaborate in detail in the scope of this paper.

[6]http://www.omg.org/spec/MARTE

[7]http://east-adl.info

VII. Conclusion

In this paper, we presented a layered (meta-)model for the automated integration of automotive software systems which we developed in the scope of the collaborative research project CCC. It ranges from a platform- and implementation-independent modelling of the functional architecture to a complete software model of its implementation for the target platform. This is intended to replace the traditional V-model design process such that the integration and testing can be performed after initial deployment and thereby allow in-field updates of automotive systems. Similar to the importance of the engineers' experience, this procedure relies on the fact that certain decisions must be made early in the design without knowing every detail of its results. In the scope of this paper, we presented the framework that ensures the soundness of these decisions by admission tests and potential design iterations whereas we will address the efficiency of this framework in future work.

Acknowledgement

This work was funded as part of the DFG Research Unit *Controlling Concurrent Change*, funding number FOR 1800.

References

[1] A. Iwai and M. Aoyama, "Automotive cloud service systems based on service-oriented architecture and its evaluation," in *Conf. on Cloud Computing*, 2011.

[2] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," in *Conf. on Dependable Systems and Networks Workshop (DSN-W)*, 2016.

[3] P. Bergmiller, *Towards Functional Safety in Drive-by-Wire Vehicles*. Springer, 2015.

[4] C. M. Kirsch and A. Sokolova, *The Logical Execution Time Paradigm*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[5] J. Schlatow, M. Moestl, and R. Ernst, "An Extensible Autonomous Reconfiguration Framework for Complex Component-Based Embedded Systems," in *Intern. Conf. on Autonomic Computing (ICAC)*, 2015.

[6] AUTOSAR, "Specification of Timing Extensions, Release 4.3," http://www.autosar.org/, 2016.

[7] R. Henia, L. Rioux, N. Sordon, G.-E. Garcia, and M. Panunzio, "Integrating Formal Timing Analysis in the Real-Time Software Development Process," in *WOSP'15*, 2015.

[8] J. Schlatow and R. Ernst, "Response-Time Analysis for Task Chains in Communicating Threads," in *22nd IEEE Real-Time Embedded Technology and Applications Symposium (RTAS 2016)*, 2016.

[9] S. Holthusen, S. Quinton, I. Schaefer, J. Schlatow, and M. Wegner, "Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates," in *Workshop on Pre- and Post-Deployment Verification Techniques (PrePost)*, Jun. 2016.

[10] J. Schlatow, M. Möstl, R. Ernst, M. Nolte, I. Jatzkowski, M. Maurer, C. Herber, and A. Herkersdorf, "Self-awareness in autonomous automotive systems," in *Design, Automation and Test in Europe*, 2017.

[11] A. Reschka, G. Bagschik, S. Ulbrich, M. Nolte, and M. Maurer, "Ability and skill graphs for system modeling, online monitoring, and decision support for vehicle guidance systems," in *Intelligent Vehicles Symposium (IV)*, 2015.

[12] M. Moestl and R. Ernst, "Handling complex dependencies in system design," in *Design, Automation Test in Europe (DATE)*, 2016.

[13] H. Lawson, S. Mubeen, A. Bucaioni, J. Mäki-Turja, J. Lundbäck, M. Gålnander, K.-L. Lundbäck, and M. Sjödin, "Provisioning of deterministic and non-deterministic services for vehicles: The Rubus Approach," in *Workshop on Critical Automotive Applications: Robustness & Safety*, 2016.

https://doi.org/10.24355/dbbs.084-201803221525

APPENDIX A
CASE STUDY

In this part of this paper, we perform a case study by conducting our automated integration process on a use case from the CCC project: an inertial navigation system (INS) that is augmented by environment perception (see Figure 2). More precisely, this use case incorporates two functionalities:

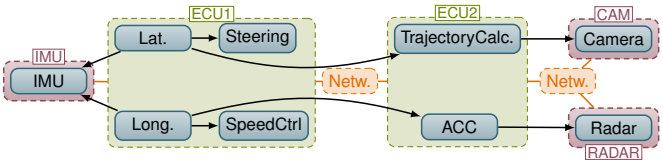


Figure 2. The inertial navigation system (INS) with environment perception.

lateral and longitudinal guidance of a road vehicle. The INS is implemented on two ECUs and dedicated hardware components that host camera, radar and IMU. ECU1 is directly connected to the IMU, which provides inertial measurements from three gyroscopes and three accelerometers. The camera, radar and both ECUs are furthermore connected to a switched Ethernet network. Figure 2 depicts the functional architecture and its predefined mapping to the target platform. Longitudinal guidance controls the vehicle speed based on the measured acceleration and speed provided by the IMU. Additionally, it includes the target velocity provided by a radar-based adaptive cruise control (ACC). Lateral guidance uses the yaw rate provided by the IMU in order to control and correct the heading. It also incorporates a reference trajectory which is calculated by a camera-based lane detection. Note that longitudinal guidance is essential for basic driving assistance systems such as ACC whereas lateral guidance gains importance when it comes to more advanced assistance systems (e.g. lane keeping) or automated driving.

Our goal is therefore to integrate both functionalities at different times on our experimental vehicle which performs the previously presented automated integration process by itself. The following part is structured as follows: We first summarise the essential aspects of the CCC framework before we explain the particular integration steps for the INS example.

A. Framework

This section gives a more detailed account of the system architecture developed in the scope of CCC. This architecture is particularly tailored for enabling in-field software updates of critical systems. For this purpose it comprises two segregated domains: model and execution domain. The execution domain hosts the actual functionality of the vehicle whereas the model domain takes control over the execution domain by changing its configuration in consequence of a software update. As shown in Figure 3, the execution domain comprises the operating system (OS) and run-time environment on which the software components are executed. It is further augmented by monitoring and shaping mechanisms, which enable the observation and enforcement of modelled behaviour

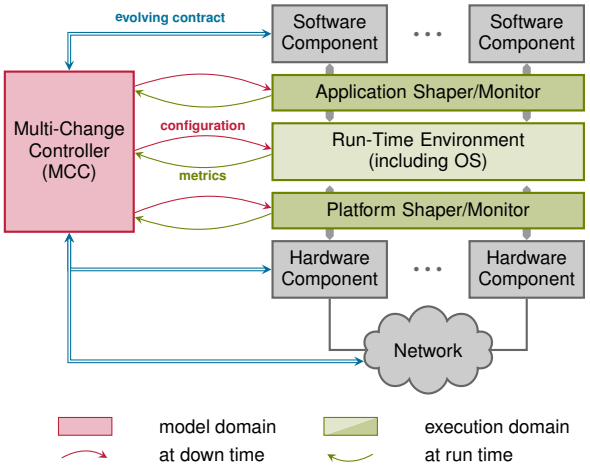


Figure 3. CCC system architecture

respectively. The model domain is implemented by the Multi-Change Controller (MCC), which equips the system with in-field update capabilities. More precisely, it applies model-based methods to find a system configuration (i.e. a G_{inst}) that satisfies all requirements such that the new configuration can be applied to the execution domain in a safe state (e.g. at function down time). Note that the MCC operates on an execution budget to not interfere with the execution domain.

Algorithm 1 sketches the integration algorithm that implements the three model transformations from Section III that provide a G_{inst}. Each transformation starts by creating a copy of the current graph. The first transformation from the functional architecture to the communication architecture is implemented by inserting proxies for arcs that cannot be implemented by direct communication. The second transformation from the communication architecture to the component architecture is implemented by selecting and replacing every node with a component pattern and by inserting protocol stacks for every arc that connects incompatible interfaces. Moreover, it will be checked whether any service has more connections than its maximum client limit in which case a multiplexer is inserted. The third transformation from the component architecture to the component instantiation consists in merging duplicate components where possible. This is implemented by the REDUCE procedure, which iterates the nodes in reverse topological order and merges duplicate components that share the same successors, i.e. only if they connect to the same services. Note that we omitted parametrisation of software components (e.g. scheduling parameters, LETs) in this framework as we address these in separate analysis engines.

B. Step-by-Step Automated Integration of the INS

In this section, we first give a more detailed account of how Algorithm 1 is applied to the longitudinal guidance. In the second part, we have a look at the update procedure that adds the lateral guidance to the INS.

Figure 4 illustrates the integration procedure of longitudinal guidance starting from G_{func}. For simplicity, we omitted the

https://doi.org/10.24355/dbbs.084-201803221525

Algorithm 1 Integration algorithm

```
1: procedure INTEGRATE(G_func)
2:     G_comm ← G_func                                  ▷ copy graph
3:     for all arc in G_comm do
4:         if ¬REACHABLE(arc) then
5:             INSERT_PROXY(G_comm, arc)
6:         end if
7:     end for
8:     G_comp ← G_comm                                  ▷ copy graph
9:     for all node in G_comp do
10:        SELECT_PATTERN(G_comp, node)
11:    end for
12:    for all arc in G_comp do
13:        if ¬COMPATIBLE(arc) then
14:            INSERT_PROTOCOLSTACK(G_comp, arc)
15:        end if
16:    end for
17:    for all arc in G_comp do
18:        #clients ← arc.target.connections
19:        limit ← arc.target.max_clients
20:        if #clients > limit then
21:            INSERT_MUX(G_comp, arc)
22:        end if
23:    end for
24:    G_inst ← G_comp                                  ▷ copy graph
25:    REDUCE(G_inst)
26:    return G_inst
27: end procedure
28: procedure REDUCE(G_inst)
29:    for all node in REVERSE_TOPOLOGICAL(G_inst) do
30:        if ∃ duplicate dup of node then
31:            if node.successors == dup.successors then
32:                MERGE(dup, node)
33:            end if
34:        end if
35:    end for
36: end procedure
```

mapping from software to hardware components for G_{func} and G_{comm}. Note that IMU and Radar are implemented on dedicated hardware components which are not managed by the MCC. After the first transformation (into G_{comm}), the unreachable arc (orange) has been replaced with a proxy.

The second transformation (to G_{comp}) starts with selecting transformation patterns. Here, only the proxy is replaced with more than one component, similar to the example in Section III. We also identified two incompatible arcs: $(Long., IMU)$ and $(ACC, Radar)$, for which the *IMU drv* and *NIC* have been inserted as the IMU can be interfaced by its device driver and the Radar via the Network. Note that the NIC is already part of the proxy pattern such that a muxer component was also inserted. As there are no duplicate components in G_{comp}, G_{inst} is identical.

Now, we add the lateral guidance to the INS following Algorithm 2. The $G_{inst,new}$ looks similar to the one for longitudinal guidance ($G_{inst,old}$) such that combing both results in duplicates of *IMU drv*, *LET rx*, *LET tx*, *muxer* and *NIC*. Reverse topological sorting (i.e. from service providers to clients) results in the following order:

1) ***IMU drv***, ***NIC1***, ***NIC2***, *SpeedCtrl*, *Steering*
2) *Long.*, *Lat.*, ***LET rx***, ***muxer***
3) ***LET tx***, *ACC*, *TrajectoryCalc*

By iterating the duplicates (bold) in this order, we start with the *NIC1*, *NIC2* and *IMU drv* components, which can be merged with their duplicates because they do not have any successor. Next, *LET rx*, *muxer* can be merged with their duplicates because they have the same successors (i.e. the same instances of *NIC1* and *NIC2* respectively). Last, *LET tx* is merged with its duplicate because both are connected to the same *muxer*. The resulting INS is depicted by Figure 5.

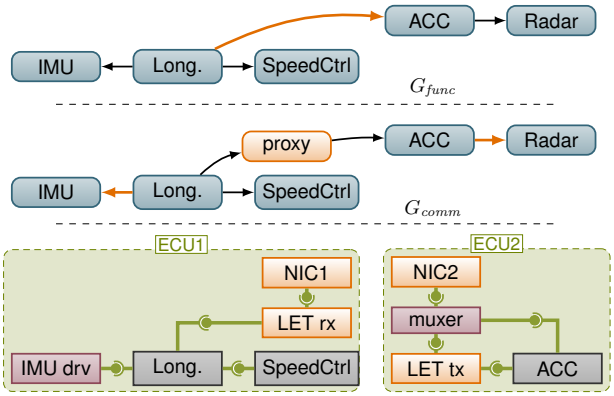


Figure 4. Steps for longitudinal guidance (G_{func}, G_{comm} and G_{comp})

Algorithm 2 Update algorithm

```
1: procedure UPDATE(G_inst,old, G_func,new)
2:     G_inst,new ←INTEGRATE(G_func,new)
3:     G_inst ← G_inst,old + G_inst,new
4:     REDUCE(G_inst)
5:     return G_inst
6: end procedure
```

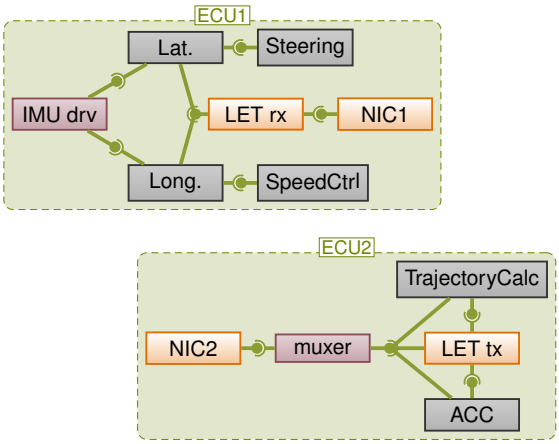


Figure 5. G_{inst} of the complete INS

https://doi.org/10.24355/dbbs.084-201803221525